FIFO Library (version 1.0) documentation

July 2000
Xavier PALLOT
National Institute of Standards and Technology

Contents

Introduction

The FIFO library provides a First-In-First-Out, that is a queue service for OPNET. The main features of this library are the multi-type data handling and the multiplexing services. The aim of this report is to describe this FIFO library. In the first part, which is intended for new users, is a general description of the library. The second part contains a specification or detailed description of each function. The third and last part is intended for developers who would like to add new behaviors to the library, thus it describes the memory structure of the FIFO.

General Description

Why this library?

OPNET provides two types of queues. First, the *queue* module (see the queue package in the OPNET documentation) provides queue multiplexing through the concept of sub-queue, but cannot contain data types other than packets. The second one is the list sub-package (see the OPNET programming package), which provides multi-type data handling, but no multiplexing.

The primary motivation to create this library was the need of a queue with the same behavior as the *queue* module, but containing events instead packets. Thus, the idea was born to create a new library that would be the marriage of both preexisting OPNET packages, that is, a queue providing both the multiplexing and the multi-types data handling services.

However, even if this new queue package seems to provide most of the services required for this kind of object, some users may need to add new behaviors. That is why the last part of this report gives a detailed description of the queue structure, which should be useful for developers.

What is the multiplexing service?

With multiplexing the modeler is not limited to only one queue but has different queues, called sub-queues, available to store and extract data. The only limitation, in terms of the number of queues, comes from the memory: as long as there are enough places to store data it can be added in a new sub-queue.

Since the multiplexing service is not always useful, it can be activated or deactivated. In fact, if the modeler wants to use this service he must use the set of functions *fifo_putInMultiplex(...)* (see p.4) and *fifo_getInMultiplex(...)* (see p.5). Through the first function data can be stored in a sub-queue that is specified by giving its number. In the same way the second function extracts the first data contained in a sub-queue identified by its number. Another function, called *fifo_getFirstInMultiplex(...)* (see p.5), gives the opportunity to extract the oldest data contained in all the sub-queues.

If the modeler does not wish to use this multiplexing service, another set of functions has been created: *fifo_getIn(...)* (see p.5) and *fifo_putIn(...)* (see p.6). In these

function, there is only one queue to manage; it is not necessary to specify any sub-queue number to store or to extract some data, respectively by calling the *fifo_getIn(...)* and the *fifo_putIn(...)* function. Since it is possible to use both sets of functions in the same time, the modeler should be careful. Read the function specifications to understand what could happen (see p.4-6) in this case.

What is the multi-type data handling service?

Multi-type data handling permits storage of any type of data in the FIFO, even in the same sub-queue. In fact the FIFO can contain any kind of data pointers. Thus to store some data, it is only necessary to provide a pointer to it in the arguments of the corresponding function: *fifo_putInMultiplex(...)* (see p.4) or *fifo_putIn(...)* (see p.5) depending on whether the multiplexing service is being used.

It is very important to note that the FIFO does not know the type of the data that are referenced by a pointer. For this reason, the library cannot provide a very "powerful" *fifo_print(...)* (see p.6) function. Moreover, the user must do a casting when the data is returned, since both FIFO functions *fifo_getInMultiplex(...)* (see p.5) and *fifo_getInFifo(...)* (see p.6) return a pointer without type. Thus the type of data referenced by the pointer must be specified before performing any manipulation on it.

Here is an example of storage & extraction of an integer:

```
{
...
sFifo myFifo;
int a=5;
int* b;
...
// give a pointer referencing the data => "&a"
putInFifo(&myFifo, &a);
...
// specify the type of data via a casting => (int*)
b=(int*)getInFifo(&myFifo);
...
}
```

How to "include" the library in your process model

Copy the three files *fifo.h, fifo.ex.c* and *fifo.s1.ex.o* into the working directory. The first file contains the header of the library, the second one its source code, and the third its compiled code. Then add in the header block of the process requiring the FIFO library the *#include "fifo.h"* compilation command so that the process knows the definition of the FIFO's structures and functions.

Finally, specify to OPNET that the *fifo* library is going to be used. This operation is done in the "File / Declare External Files …" menu, by given the status *included* to the entry *fifo*. If the *fifo* entry does not appear in the menu, try the "File / Refresh Model

Directories" command. If the entry still does not appear, that means that the working directory is not defined correctly in the "Edit / Preferences / mod_dirs" option menu.

Specifications

This second section includes the description of every function in the FIFO library. Each description is divided in two parts: a header and a behavior description. Both are very useful for using a function correctly.

Function fifo_init

Header:
void initFifo(sFifo *fifo)
sFifo fifo: a pointer to the FIFO structure to initialize

Description:
This function initializes a FIFO structure. This function must be called before using any FIFO declared as static variable. Note that this function is called automatically when building a dynamic FIFO through the *fifo_new(...)* function.

Function fifo_new

Header:
sFifo* newFifo( )
Return: a pointer to the new dynamic FIFO structure

Description:
This function creates, initializes and returns a new FIFO structure located in dynamic memory. Note that the *fifo_destroy (...)* function must be called in order to remove this structure from memory.

Function fifo_putInMultiplex

Header:
int putInFifoMultiplex (sFifo* fifo, void* data, int fifoNumber)
sFifo* fifo: a pointer to the FIFO
void*data: a pointer to the data to store in one of the sub-queue of the FIFO
fifoNumber: the number of the FIFO sub-queue in which the data must be stored
Return: 1 if success, 0 otherwise

Description:
This function adds the pointer referencing the data at the end of the sub-queue identified by the *fifoNumber* parameter. It returns 1 if the operation is successful, i.e. if there is still enough place to store the pointer, 0 otherwise.

Function fifo_putIn

Header:
int putInFifo(sFifo* fifo, void* data)
sFifo* fifo: a pointer to the FIFO
void* data: a pointer to the data to add in the FIFO
Return: 1 if success, 0 otherwise

Description:
This function adds the pointer referencing the data at the end of the queue. It returns 1 if the operation is successful, i.e. if there is still enough place to store the pointer, 0 otherwise.

Caution: be aware that if you are using this function for a FIFO on which you have already used the multiplexing service (by calling the *fifo_putInMultiplex (...)* function), the data pointer will be stored in the default sub-queue number 0.

Function fifo_getInMultiplex

Header:
void* getInFifoMultiplex(sFifo* fifo, int fifoNumber)
sFifo* fifo: a pointer to the FIFO
int fifoNumber: the number of the FIFO sub-queue from which you want to extract the data
Return: a pointer to the extracted data if success, OPC_NIL otherwise

Description:
This function extracts and returns a pointer to the oldest data contained in the sub-queue identified by the *fifoNumber* parameter. Note that if no data is found the OPC_NIL constant is returned.

Function fifo_getFirstInMultiplex

Header:
void* getFirstInFifoMultiplex(sFifo *fifo, int *fifoNumber)
sFifo fifo: a pointer to the FIFO
int fifoNumber: is used to return the sub-queue number of the extracted data
Return: a pointer to the extracted data if success, OPC_NIL otherwise

Description:
This function returns a pointer to the oldest data stored in a FIFO using the multiplexing service. That is, it extracts the oldest data without caring about the sub-queue number. The function also returns, via the *fifoNumber* parameter, the sub-queue number from which it got the data. Note that if no data is found the OPC_NIL constant is returned.

Function fifo_getIn

Header:
void* getInFifo(sFifo *fifo)
sFifo* fifo: a pointer to the FIFO
Return: a pointer to the extracted data if success, OPC_NIL otherwise

Description:
This function extracts and returns a pointer to the oldest data contained in the queue. Note that if no data is found the OPC_NIL constant is returned.

Caution: be aware that if you are using this function for a FIFO on which you have already used the multiplexing service (by calling the *fifo_putInMultiplex (...)* function), the data pointer will be extracted from the default sub-queue number 0.

Function fifo_destroy

Header:
void destroyFifo(sFifo* fifo)
*sFifo fifo: a pointer to the FIFO structure to remove from dynamic memory

Description:
This function removes a FIFO structure, created via the *fifo_new (...)* function, from the dynamic memory. Note that it also destroys all the objects and data, which are contained in the FIFO.

Caution: never use this function for a FIFO declared as static variable.

Function fifo_print

Header:
void printFifo(sFifo fifo)
sFifo fifo: the FIFO to display

Description:
This function displays a FIFO on the computer screen. However, since the data type is unknown to the FIFO structure, it is impossible to display the data itself. Thus this function display only the address of each data pointer contains in the queue.

Structure description

In this section, it will be explained how the FIFO works, i.e. how its memory structure allows it to provide both multiplexing and multi-type data handling services.

Structure overview

First of all, it is important to notice that the FIFO is one and only one queue in memory, even if multiple sub-queues are available. In fact to provide the multiplexing service, a number is associated with each data stored in the queue. This number represents the sub-queue number in which the data is inserted, and is provided by the user when he calls the *fifo_putInMultiplex(...)* function (see p.4). Moreover, as it is said in the second section, the default sub-queue is the sub-queue number 0. Therefore, when the modeler adds some data by calling the *fifo_putIn(...)* (see p.5) function, the number associated with them in memory is 0.

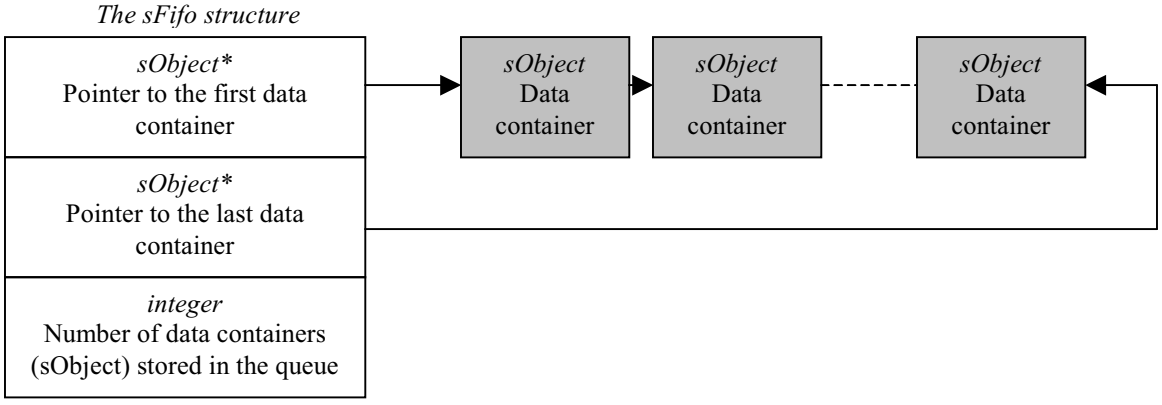
One queue means that the memory structure of the FIFO is very classical. In fact, only the objects containing the data, thus playing the role of container, have some particularities in order to provide the wanted services.

The FIFO structure: sFifo

Here is the definition of the queue structure:

```
typedef struct
    {
    // number of objects contained in the "FIFO"
    int nbrObjects;
    // pointer to the first object of the "FIFO"
    sObject* firstObject;
    // pointer to the last object of the "FIFO"
    sObject* lastObject;
    } sFifo;
```

This queue structure is very classical, since it contains a pointer to the first object (for data extraction), a pointer to the last object (for data addition), and an integer to store the number of objects currently contained in the queue.
The following schematic representation of the queue is consequently very usual:



The data container structure: sObject

As it is written in the scheme above, the queue data containers are defined by the *sObject* structure. Here is the definition of this structure:

```
struct sObject;
typedef struct
        {
        // pointer to data of any types
        void* data;
        // FIFO number for multiplexage
        int fifoNumber;
        // pointer to the next Object of the "FIFO"
        struct sObject* next;
        } sObject;
```

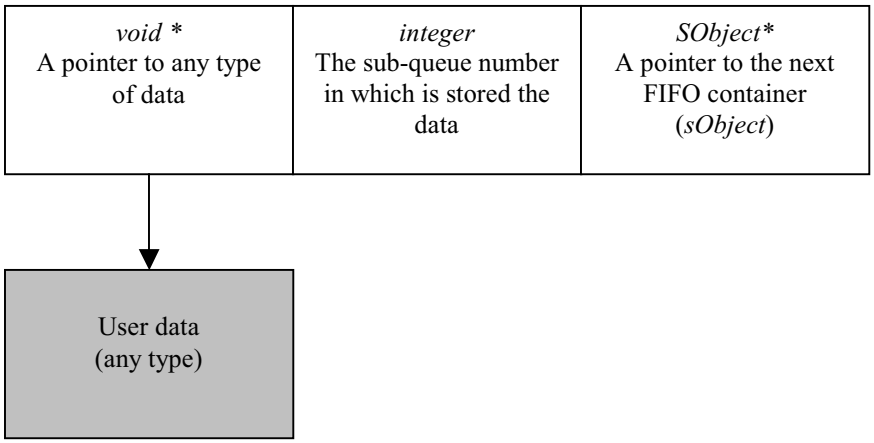
This structure includes some particularities in order to provide the specific services of the FIFO. On the one hand, as is described at the beginning of this section, a number (*fifoNumber*) is associated with every data. This number allows the multiplexing services by indicating the sub-queue number of the data.

On the second hand, a pointer (*void * data*) without any type is used to reference the user's data. This un-type pointer is the base of the multi-type handling service, since no types of data are expected from the user. However, it also means that the FIFO doesn't know the type of data that it stores, which has some consequences: the limitation of the *fifo_print(...)* function (see p.6), and the obligation for the modeler to do a casting when he gets back his data (see p.3).

The last field of the *sObject* structure (*next*) is very usual, since it is a pointer to the next data container of the queue. This field is equal to the OPNET *OPC_NIL* constant if the *sObject* is the last one of the queue.

Finally here is the schematic representation of the *sObject*, that is the queue container structure:

The sObject structure



Conclusion

This OPNET queue library, called FIFO, may be useful for many OPNET modelers via its both multiplexing and multi-type data handling services. Many functions have been designed in order to provide a simple to use and a powerful tool to its users. This documentation explains how to use this library through a general description and a complete specification of its functions.

Moreover, since neither the world nor this library could be perfect, some developers may wish to add some new behaviors to the present FIFO. The last section, which describes precisely the queue structure, has been written for them. It could be also interesting for modelers, who would like to understand precisely how the library works, to read it.

Finally, I hope that this FIFO library will be helpful for many OPNET modelers in order to gain time, and to build great process models.